EEE 443 - Neural Networks
Assignment 1

Ege Ozan Özyedek
21703374

October 19, 2020

1 Question 1

We are asked to find the probability distribution of the weights $P(W)$ given the following optimization problem

$$W_{MAP} = \arg\min_{W} \sum_{n} \left(y^n - h\left(x^n, W\right)\right)^2 + \beta \sum_{i} w_i^2$$

This can be simplified by using the 2-norm (the Euclidian norm)

$$W_{\text{MAP}} = \arg\max_{W} \left(\|Y - h(X, W)\|^2 + \beta \|W\|^2 \right)$$

The general Maximum A Posteriori formula is as follows

$$W_{\text{MAP}} = \arg\max_{W} P(W \mid X, Y) = \arg\max_{W} \frac{P(W)P(X, Y \mid W)}{P(X, Y)}$$

The probability $P(X, Y)$ has no effect on the equation since it is not dependent on W, hence can be eliminated. Furthermore, we can apply the log function to the equation since the natural logarithm is a monotone function and does not change the output of the MAP estimate.

$$W_{\text{MAP}} = \arg\max_{W} P(W)P(X, Y \mid W)$$

$$W_{\text{MAP}} = \arg\max_{W} \; log(P(W)P(X, Y \mid W))$$

$$W_{\text{MAP}} = \arg\max_{W} \left[log(P(X, Y \mid W)) + log(P(W))\right]$$

The equation found is similar to the optimization problem given. We will now factor out the two probabilities by looking at their dependencies. It is clear that the first norm is dependent on X and Y for a given W. Hence we can say that

$$log(P(X, Y \mid W)) = \|Y - h(X, W)\|^2$$

Also evidently, the second norm is only dependent on W, and does not need a given X or Y. Hence we find that

$$log(P(W)) = \beta \|W\|^2$$

From here, the only thing we need to do is to single out the probability $P(W)$ by taking the exponential power of both sides.

$$e^{log(P(W))} = e^{\beta \|W\|^2}$$

Hence we can find the final forms of the equation as follows

$$P(W) = e^{\beta \|W\|^2}$$

$$P(W) = e^{\beta \sum_i w_i^2}$$

$$P(W) = \prod_i e^{\beta w_i^2}$$

2 Question 2

a) We are required to implement the following logic function using a hidden layer with 4 neurons.

$$(X_1 \vee \neg X_2) \oplus (\neg X_3 \vee \neg X_4)$$

Using the fact that $a \oplus b = (a \cdot \overline{b}) + (\overline{a} \cdot b)$ the above expression can be simplified to find the expressions of each of the 4 hidden layers. Starting by using De Morgan's law, we find

$$((X_1 \vee \neg X_2) \wedge \neg(\neg X_3 \vee \neg X_4)) \vee (\neg(X_1 \vee \neg X_2) \wedge (\neg X_3 \vee \neg X_4))$$

$$((X_1 \vee \neg X_2) \wedge (X_3 \wedge X_4)) \vee ((\neg X_1 \wedge X_2) \wedge (\neg X_3 \vee \neg X_4))$$

Using the distributive property of $\wedge$ find the final expression as

$$(X_1 \wedge X_3 \wedge X_4) \vee (\neg X_2 \wedge X_3 \wedge X_4) \vee (\neg X_1 \wedge X_2 \wedge \neg X_3) \vee (\neg X_1 \wedge X_2 \wedge \neg X_4)$$

The above equation is now in the form of 2 blocks, one AND and one OR. The 4 neurons in the hidden layer as well as the output expression can be labeled as seen below.

$$h_1 = X_1 \wedge X_3 \wedge X_4$$

$$h_2 = \neg X_2 \wedge X_3 \wedge X_4$$

$$h_3 = \neg X_1 \wedge X_2 \wedge \neg X_3$$

$$h_4 = \neg X_1 \wedge X_2 \wedge \neg X_4$$

$$o = h_1 \vee h_2 \vee h_3 \vee h_4$$

Since we have a clear expression of all elements in the network, we can write out the truth tables for the hidden layer neurons and the output. These truth tables will give us the analatical derivation of inequalities which determine the weights. The activation function will be denoted as $f(v)$ and is the unipolar step function. The weight vector can be defined as seen below. θ represents the biasing term for each neuron and w_i represents the weights affiliated with the i^th neuron in the network (w_5 represents the weights for the output layer).

$$W_{in} = \begin{bmatrix} w_1^T \\ w_2^T \\ w_3^T \\ w_4^T \end{bmatrix} = \begin{bmatrix} w_{11} & w_{12} & w_{13} & w_{14} & \theta_1 \\ w_{21} & w_{22} & w_{23} & w_{24} & \theta_2 \\ w_{31} & w_{32} & w_{33} & w_{34} & \theta_3 \\ w_{41} & w_{42} & w_{43} & w_{44} & \theta_4 \end{bmatrix}$$

$$W_h = \begin{bmatrix} w_5^T \end{bmatrix} = \begin{bmatrix} w_{51} & w_{52} & w_{53} & w_{54} & \theta_5 \end{bmatrix}$$

Table 1: Truth Table and Inequalities for $\mathbf{h_1 = X_1 \wedge X_3 \wedge X_4}$.

Since h_1 does not take X_2 as an input, it is not included in the table. Also because of this, its weight is pre-determined as $w_{12} = 0$.

X_1	X_3	X_4	h_1	Equation	Inequality
0	0	0	0	$f(-\theta_1) = 0$	$\theta_1 > 0$
0	0	1	0	$f(w_{14} - \theta_1) = 0$	$w_{14} - \theta_1 < 0$
0	1	0	0	$f(w_{13} - \theta_1) = 0$	$w_{13} - \theta_1 < 0$
0	1	1	0	$f(w_{13} + w_{14} - \theta_1) = 0$	$w_{13} + w_{14} - \theta_1 < 0$
1	0	0	0	$f(w_{11} - \theta_1) = 0$	$w_{11} - \theta_1 < 0$
1	0	1	0	$f(w_{11} + w_{14} - \theta_1) = 0$	$w_{11} + w_{14} - \theta_1 < 0$
1	1	0	0	$f(w_{11} + w_{13} - \theta_1) = 0$	$w_{11} + w_{13} - \theta_1 < 0$
1	1	1	1	$f(w_{11} + w_{13} + w_{14} - \theta_1) = 1$	$w_{11} + w_{13} + w_{14} - \theta_1 > 0$

Table 2: Truth Table and Inequalities for $\mathbf{h_2 = \neg X_2 \wedge X_3 \wedge X_4}$.

Since h_2 does not take X_1 as an input, it is not included in the table. Also because of this, its weight is pre-determined as $w_{21} = 0$.

X_2	X_3	X_4	h_2	Equation	Inequality
0	0	0	0	$f(-\theta_2) = 0$	$\theta_2 > 0$
0	0	1	0	$f(w_{24} - \theta_2) = 0$	$w_{24} - \theta_2 < 0$
0	1	0	0	$f(w_{23} - \theta_2) = 0$	$w_{23} - \theta_2 < 0$
0	1	1	1	$f(w_{23} + w_{24} - \theta_2) = 1$	$w_{23} + w_{24} - \theta_2 > 0$
1	0	0	0	$f(w_{22} - \theta_2) = 0$	$w_{22} - \theta_2 < 0$
1	0	1	0	$f(w_{22} + w_{24} - \theta_2) = 0$	$w_{22} + w_{24} - \theta_2 < 0$
1	1	0	0	$f(w_{22} + w_{23} - \theta_2) = 0$	$w_{22} + w_{23} - \theta_2 < 0$
1	1	1	0	$f(w_{22} + w_{13} + w_{14} - \theta_2) = 0$	$w_{22} + w_{23} + w_{24} - \theta_2 < 0$

Table 3: Truth Table and Inequalities for $\mathbf{h_3 = \neg X_1 \wedge X_2 \wedge \neg X_3}$.

Since h_3 does not take X_4 as an input, it is not included in the table. Also because of this, its weight is pre-determined as $w_{34} = 0$.

X_1	X_2	X_3	h_3	Equation	Inequality
0	0	0	0	$f(-\theta_3) = 0$	$\theta_3 > 0$
0	0	1	0	$f(w_{33} - \theta_3) = 0$	$w_{33} - \theta_3 < 0$
0	1	0	1	$f(w_{32} - \theta_3) = 1$	$w_{32} - \theta_3 > 0$
0	1	1	0	$f(w_{32} + w_{33} - \theta_3) = 0$	$w_{32} + w_{33} - \theta_3 < 0$
1	0	0	0	$f(w_{31} - \theta_3) = 0$	$w_{31} - \theta_3 < 0$
1	0	1	0	$f(w_{31} + w_{33} - \theta_3) = 0$	$w_{31} + w_{33} - \theta_3 < 0$
1	1	0	0	$f(w_{31} + w_{32} - \theta_3) = 0$	$w_{31} + w_{32} - \theta_3 < 0$
1	1	1	0	$f(w_{31} + w_{32} + w_{33} - \theta_3) = 0$	$w_{31} + w_{32} + w_{33} - \theta_3 < 0$

Table 4: Truth Table and Inequalities for $\mathbf{h_4 = \neg X_1 \wedge X_2 \wedge \neg X_4}$.

Since h_4 does not take X_3 as an input, it is not included in the table. Also because of this, its weight is pre-determined as $w_{43} = 0$.

X_1	X_2	X_4	h_4	Equation	Inequality
0	0	0	0	$f(-\theta_4) = 0$	$\theta_4 > 0$
0	0	1	0	$f(w_{44} - \theta_4) = 0$	$w_{44} - \theta_4 < 0$
0	1	0	1	$f(w_{42} - \theta_4) = 1$	$w_{42} - \theta_4 > 0$
0	1	1	0	$f(w_{42} + w_{44} - \theta_4) = 0$	$w_{42} + w_{44} - \theta_4 < 0$
1	0	0	0	$f(w_{41} - \theta_4) = 0$	$w_{41} - \theta_4 < 0$
1	0	1	0	$f(w_{41} + w_{44} - \theta_4) = 0$	$w_{41} + w_{44} - \theta_4 < 0$
1	1	0	0	$f(w_{41} + w_{42} - \theta_4) = 0$	$w_{41} + w_{42} - \theta_4 < 0$
1	1	1	0	$f(w_{41} + w_{42} + w_{44} - \theta_4) = 0$	$w_{41} + w_{42} + w_{44} - \theta_4 < 0$

Table 5: Truth Table and Inequalities for $\mathbf{o = h_1 \vee h_2 \vee h_3 \vee h_4}$, weight vector $\mathbf{W_h = w_5^T}$.

h_1	h_2	h_3	h_4	o	Equation	Inequality
0	0	0	0	0	$f(-\theta_5) = 0$	$\theta_5 > 0$
0	0	0	1	1	$f(w_{54} - \theta_5) = 1$	$w_{54} - \theta_5 > 0$
0	0	1	0	1	$f(w_{53} - \theta_5) = 1$	$w_{53} - \theta_5 > 0$
0	0	1	1	1	$f(w_{53} + w_{54} - \theta_5) = 1$	$w_{53} + w_{54} - \theta_5 > 0$
0	1	0	0	1	$f(w_{52} - \theta_5) = 1$	$w_{52} - \theta_5 > 0$
0	1	0	1	1	$f(w_{52} + w_{54} - \theta_5) = 1$	$w_{52} + w_{54} - \theta_5 > 0$
0	1	1	0	1	$f(w_{52} + w_{53} - \theta_5) = 1$	$w_{52} + w_{53} - \theta_5 > 0$
0	1	1	1	1	$f(w_{52} + w_{53} + w_{54} - \theta_5) = 1$	$w_{52} + w_{53} + w_{53} - \theta_5 > 0$
1	0	0	0	1	$f(w_{51} - \theta_5) = 1$	$w_{51} - \theta_5 > 0$
1	0	0	1	1	$f(w_{51} + w_{54} - \theta_5) = 1$	$w_{51} + w_{54} - \theta_5 > 0$
1	0	1	0	1	$f(w_{51} + w_{53} - \theta_5) = 1$	$w_{51} + w_{53} - \theta_5 > 0$
1	0	1	1	1	$f(w_{51} + w_{53} + w_{54} - \theta_5) = 1$	$w_{51} + w_{53} + w_{54} - \theta_5 > 0$
1	1	0	0	1	$f(w_{51} + w_{52} - \theta_5) = 1$	$w_{51} + w_{52} - \theta_5 > 0$
1	1	0	1	1	$f(w_{51} + w_{52} + w_{54} - \theta_5) = 1$	$w_{51} + w_{52} + w_{54} - \theta_5 > 0$
1	1	1	0	1	$f(w_{51} + w_{52} + w_{53} - \theta_5) = 1$	$w_{51} + w_{52} + w_{53} - \theta_5 > 0$
1	1	1	1	1	$f(w_{51} + w_{52} + w_{53} + w_{54} - \theta_5) = 1$	$w_{51} + w_{52} + w_{53} + w_{53} - \theta_5 > 0$

b) From the above inequalities , we can find the weight vectors which satisfy %100 efficiency for the neural network. The input weight matrix W_{in} which contains the weight vectors $w_i, i\epsilon 1, 2, 3, 4$, and the hidden layer weight vector W_h. Below, the weights I have acquired from the inequalities can be found.

$$W_{in} = \begin{bmatrix} w_1^T \\ w_2^T \\ w_3^T \\ w_4^T \end{bmatrix} = \begin{bmatrix} w_{11} & w_{12} & w_{13} & w_{14} & \theta_1 \\ w_{21} & w_{22} & w_{23} & w_{24} & \theta_2 \\ w_{31} & w_{32} & w_{33} & w_{34} & \theta_3 \\ w_{41} & w_{42} & w_{43} & w_{44} & \theta_4 \end{bmatrix} = \begin{bmatrix} 0.4 & 0 & 0.4 & 0.4 & 1 \\ 0 & -0.6 & 0.6 & 0.6 & 1 \\ -0.4 & 1.2 & -0.4 & 0 & 1 \\ -0.4 & 1.2 & 0 & -0.4 & 1 \end{bmatrix}$$

$$W_h = \begin{bmatrix} w_5^T \end{bmatrix} = \begin{bmatrix} w_{51} & w_{52} & w_{53} & w_{54} & \theta_5 \end{bmatrix} = \begin{bmatrix} 1 & 1 & 1 & 1 & 0.5 \end{bmatrix}$$

The accuracy for this question, as well as parts c and d were calculated by code, which can be viewed at the end of the report. To find an average of the value, the accuracy is calculated 100 times and added up to a sum for each question and the mean of this value is presented as the accuracy.

c) For this problem, small random fluctuations (from $N(0, 0.1)$) were added to the input (excluding the bias) to simulate noise. The weight vectors given above were tested with the noise added inputs. This resulted in an accuracy value of 92.5%, which is acceptable but can be improved. After inspecting the input weight vectors W_{in} chosen in the previous section, I decided that by equalizing the weights of all inputs to ± 1 I could achieve better performance. This has two interpretations. The first interpretation would be how far away the graphical interpretation of the network is from each point, by equalizing each input weight we create a higher threshold for fluctuations and noise in inputs. The other interpretation is importance. Since all inputs are equally important for the output, by equating all of them to ± 1 we even out the playing field. The weight vectors I chose previously both created an advantage for inputs (some inputs have 1.2 weights while others have 0.4) and also made the graphical interpretation far away from some outputs and closer to others. Since the thresholds (biases) θ_i's are not affected by noise, we don't have to have them at 1 or equal to the other weights. This way we also comply with the inequalities we have found in section 2.a. The hidden layer weights remain unchanged since these weights already separated equally and create equal importance for the inputs. The new weights can be found below.

$$W_{in\_updated} = \begin{bmatrix} w_1^T \\ w_2^T \\ w_3^T \\ w_4^T \end{bmatrix} = \begin{bmatrix} w_{11} & w_{12} & w_{13} & w_{14} & \theta_1 \\ w_{21} & w_{22} & w_{23} & w_{24} & \theta_2 \\ w_{31} & w_{32} & w_{33} & w_{34} & \theta_3 \\ w_{41} & w_{42} & w_{43} & w_{44} & \theta_4 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 1 & 1 & 2.5 \\ 0 & -1 & 1 & 1 & 1.5 \\ -1 & 1 & -1 & 0 & 0.5 \\ -1 & 1 & 0 & -1 & 0.5 \end{bmatrix}$$

$$W_h = \begin{bmatrix} w_5^T \end{bmatrix} = \begin{bmatrix} w_{51} & w_{52} & w_{53} & w_{54} & \theta_5 \end{bmatrix} = \begin{bmatrix} 1 & 1 & 1 & 1 & 0.5 \end{bmatrix}$$

This update gave better performance, an accuracy of 97.75%. This shows that the network is much more resistant to noise and fluctuations given equal importance to inputs (or equally separating points in the graphical interpretation).

d) In this section, 25 of the 16 cases $(25 * 16)$ that we have worked on were created as input samples for each of the 4 inputs (this creates a (400,4) sized matrix). To each of the 400 samples, a randomly generated Gaussian noise $(N(0, 0.2))$ was added (this also creates a noise matrix sized (400,4)). Then this noise matrix and the input matrix were added. To this vector, a column of -1's were added as the bias inputs (size (400,1)). This vector can be found below.

$$X = \begin{bmatrix} x_1 & x_2 & x_3 & x_4 \end{bmatrix}_{400,4}, noise = \begin{bmatrix} n_1 & n_2 & n_3 & n_4 \end{bmatrix}_{400,4}$$

$$X_{noisy} = \begin{bmatrix} x_1 + n_1 & x_2 + n_2 & x_3 + n_3 & x_4 + n_4 \end{bmatrix}_{400,4}, \theta = \begin{bmatrix} -1 \\ . \\ . \\ . \\ -1 \end{bmatrix}_{400,1}$$

$$X_{final} = \begin{bmatrix} X_{noisy} & \theta \end{bmatrix}_{400,5}$$

Using the input matrix X_{final}, accuracy was computed with the 2 different weight matrices found in parts 2.a-b and 2.c. For the weights W_in and W_h the accuracy was 86.3% (question 2.a-b). For the weights W_in and W_h the accuracy was 90.5% (question 2.c). This again supports our case that when the weights are equally separated, the network becomes more susceptible to noise, hence is more accurate in its classification.

3 Question 3

a) The required sample images and the correlation matrix can be found below

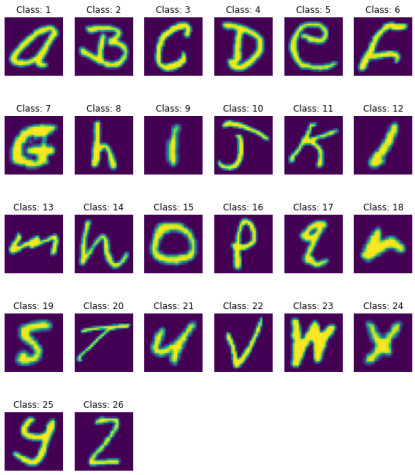


Figure 1: Sample images of each class

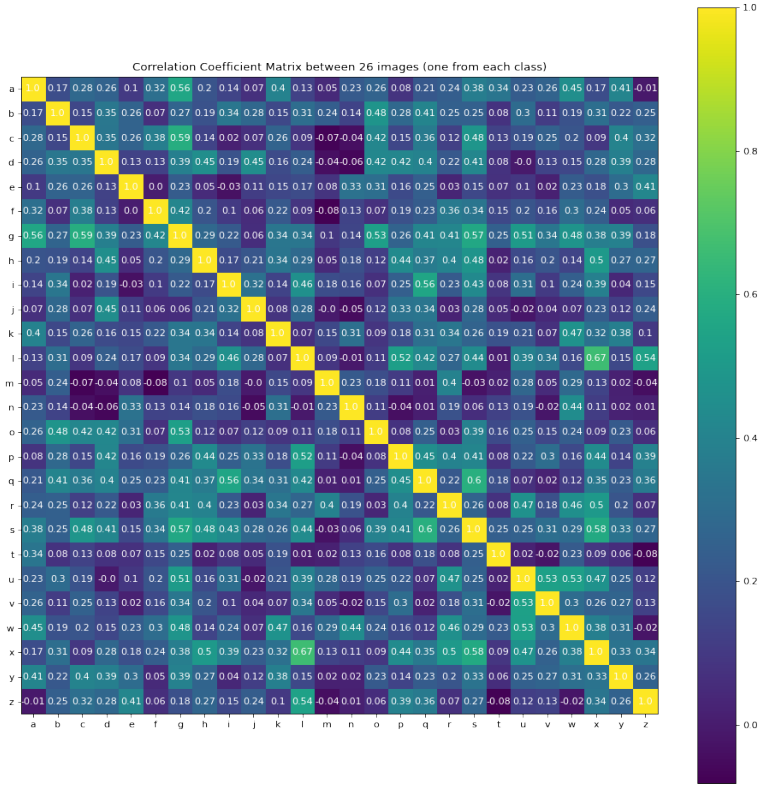


Figure 2: The correlation matrix across 26 samples in Figure 1

Observing the correlation matrix, we can come to several conclusions. Letters that contain diagonal lines (ex.: l, x, z) correlate more with each other compared to letters that have curved lines (ex.: a, b, c, o, g). For example, in the sample images l seems like one diagonal line of x, and this is true in the correlation matrix as well since the correlation value is 0.67. Other examples exits, such as a and g. In the sample images a and capital g look similar because of their round shape and the lines sticking out of this round shape. As a final example, lets examine m. The matrix shows us that the letter m is similar to n, r and w. This is expected since the letter m contains shapes of n and r, and w is similar to an upside down m. The similarity is also evident in the sample images. Hence, the correlation matrix gives us an idea of where the network might make mistakes. Some letters are more likely to be detected as the letter a for example. This information might be beneficial in improving the network in future iterations. Within class the correlation is 1, since the images are the same.

b) The visualization of weights after training with $\eta = 0.146$, which was found to be the optimal eta, can be found below.

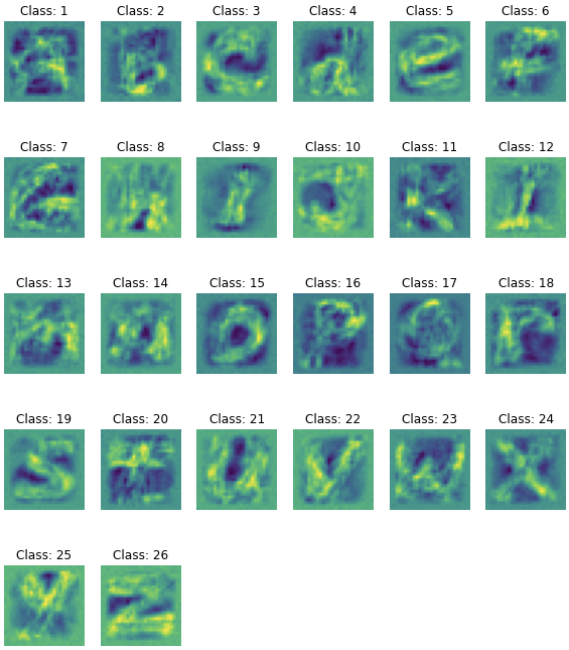


Figure 3: Visualized weight vectors for $\eta = 0.146$

It can be observed that the visualization of the weight vectors resemble the letters themselves. This expected and desired, since if the given input is not similar to the shape of the weight their sigmout output will be low and hence will not be chosen as the final determined letter. Some letters, such as Q and T look less like the letter it represents. This may be because of the correlations of

these letter to other letters (this was talked about in 3.a). Hence, these weight vectors are more "undetermined" and look blurred.

c) The MSE plots for three different η values can be observed below.

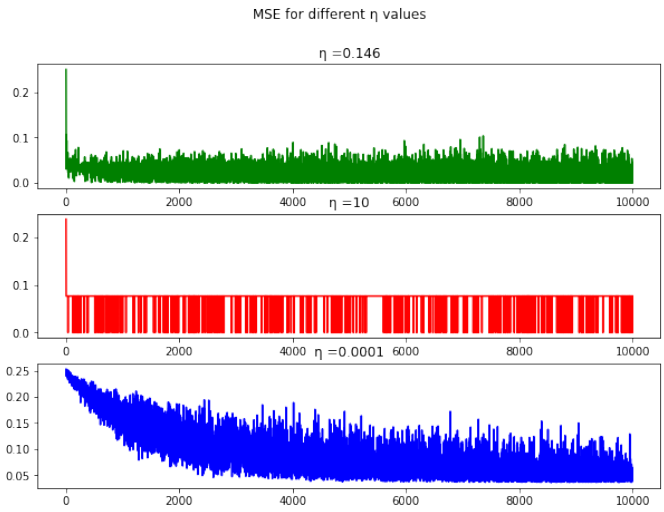


Figure 4: MSE Plots, x axis represents training samples, y axis represents the MSE

The optimal η value was chosen to be $\eta = 0.146$. This is because with this learning rate, the network has a very low minimum MSE $\simeq 8 * 10^{-11}$ and is more stable in its oscillations compared to the other values, as the graph visualizes.

For the $\eta_{high} = 10$ value, the graph shows rapid oscillations between MSE $\simeq 0$ to 0.1. This is an outcome of high learning rate, since the rate is high, the weights oscillate between two vector values. Hence, the MSE also oscillates between two points. As an added proof to this case, the weight visualizations at this learning rate only show two different pictures, and can be observed below.

The low $\eta = 0.0001$ value shows how slow the learning process becomes when the learning rate is chosen low. The MSE plot displays its slow learning, it can be seen that compared to the other two values of η which have their minimum MSE values around 0, η_{low} has its minimum eta value at around 0.05.

From this analysis we can conclude that the optimal learning rate for this problem was around the values of 0.1 to 1. However, the learning rate seems to be a variable that differs from problem to problem. As such, a high learning rate might be suitable for data that has values that differ from each other substantially and low learning rate might be used to precisely compute weight vectors on a simpler function that can be run through higher epochs.

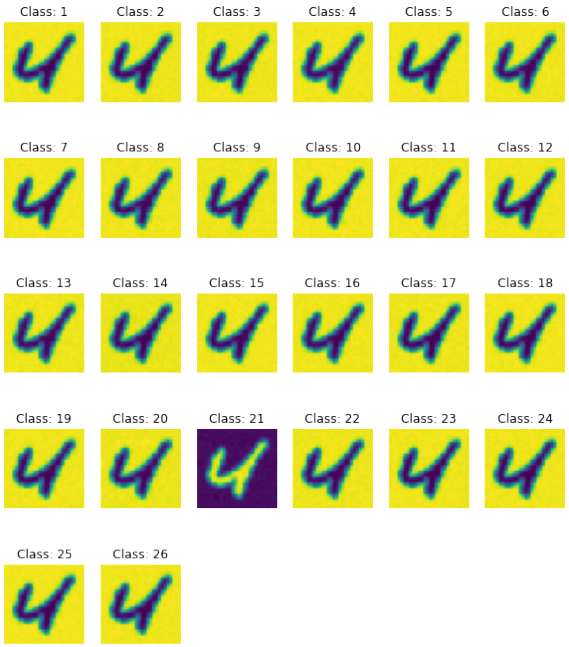


Figure 5: Visualized weight vectors for $\eta = 10$

d) The below figure show the console output of the classification accuracy (which is deemed as performance) over training and test data for three of the mentioned η values. The accuracy is computed by using all samples in both datasets (5200 samples for training data and 1300 samples for test data). The specifics on the computation of the accuracy can be found at the end of this report, in the code section.

```
Results for η = 0.146
Train accuracy:  65.07692307692308  %
Test accuracy:  60.76923076923077  %
Results for η = 10
Train accuracy:  3.8461538461538463  %
Test accuracy:  3.8461538461538463  %
Results for η = 0.0001
Train accuracy:  18.98076923076923  %
Test accuracy:  17.46153846153846  %
```

Figure 6: Console output for the accuracy performance of different learning rates

These performances comply with our discussions in the previous sections. The optimal η^* value has the best test performance with 61%. This expected and good for the problem and conditions at hand (since this question is on a

single layer NN and the cost function is perhaps not the most optimal one, this accuracy is acceptable). The high η_{high} value has a very low accuracy of 3.85% and this accuracy is the same between the test and train samples. Taking this and the visualization of the weights for this learning rate in mind, it can be understood that with such a high learning rate this network only can classify a single letter, hence the low performance. The low η_{low} value has low accuracy but the reasoning behind it is that it couldn't reach a high accuracy since it was upgrading the weights too slowly.

4 Question 4

This Jupyter Notebook acts as learning step for 2-layered neural networks. First, a "toy" case is created to test out the features of the 2-layer neural network class $TwoLayerNet$. Then, forward pass is coded sequentially. Forward pass starts with the computation of the "scores". The scores is the output of the network before the activation function. $TwoLayerNet$ class uses the ReLU function to compute the activation output of the hidden layers, which is then used to find the scores. The ReLU function can be observed below.

$$ReLU(X) = max(0, X)$$

The ReLU function is a non-linear activation function that holds many advantages over other activation functions such as sigmoid because of its linear-like nature. Unlike sigmoid where the gradient gets smaller as the input gets larger, ReLU has constant gradient which results in faster learning. Using ReLU, the hidden layer activation is found. Using the hidden layer activation, the scores are found. The equations of these steps can be seen below.

$$o_h = ReLU(Wx + b_1)$$

$$scores = o_h \cdot x + b_2$$

Next, the loss is computed. The output activation function is a softmax function. This function is a widely used output activation function that gives probabilities of classes occurring. The softmax function can be observed below.

$$softmax(x)_i = \frac{e^{x_i}}{\sum_j e^{x_j})}$$

The loss equation is one that is affiliated with the softmax function. Its called the cross-entropy loss. It is divided into two parts as the data loss and the regularization loss. The loss is computed in the notebook using this equation. After the loss is computed, the derivatives for the gradient descent are computed. Then Stochastic Gradient Descent is used in place of regular gradient descent to train the network. These steps are similar to Question 3; however with different functions and gradients as noted above. After the "toy" case, the notebook loads data of CIFAR-10, which contains multiple images from multiple classes such as different animals (cats, dogs, ...) and different vehicles (airplanes, ships, ...). SGD is used to train the neural network; however the accuracy of the network is around 29%, which is very low. To understand the reason behind this outcome, the plots of loss over iteration and accuracy of test and train data over epoch are visualized. Additionally, similar to what was done in Question 3, the first layer weights are visualized as well. Below the visualization are some ideas about what might be the problem. Parameters such as the learning rate, hidden unit number and regularization strength affect

the test accuracy, and hence fine-tuning these parameters might give better performance. The notebook has the best validation accuracy as 48.6% which yields below test accuracy and has the below parameters
$Test\ Accuracy = 48.2\%$
$\eta = 0.001$
$hidden\ units = 100$
$regularization\ strength = 0.75$
I managed to get a high test accuracy by using the below parameters
$Test\ Accuracy = 50.06\%$
$\eta = 0.0015$
$hidden\ units = 75$
$regularization\ strength = 0.55$
I visualized the weights from the best case I found, which can be observed below. Similar to Question 3, the weights resemble the classes that they test, for example some of the below weights resemble animals like horses. .

Figure 7: Weight visualization for Question 4

Jupyter Notebooks

Question 2 and Question 3

The following pages in this subsection contain the code written for Question 2 and Question 3

1 Dependencies

[3]:
```
import numpy as np
import h5py
import matplotlib.animation as animation
import matplotlib.pyplot as plt
import string
import sys
import math
```

2 Question 2

[4]:
```
def question_2():
    X = np.array([
                [0, 0, 0, 0, -1],
                [0,0,0,1,-1],
                [0,0,1,0,-1],
                [0,0,1,1,-1],
                [0,1,0,0,-1],
                [0,1,0,1,-1],
                [0,1,1,0,-1],
                [0,1,1,1,-1],
                [1,0,0,0,-1],
                [1,0,0,1,-1],
                [1,0,1,0,-1],
                [1,0,1,1,-1],
                [1,1,0,0,-1],
                [1,1,0,1,-1],
                [1,1,1,0,-1],
                [1,1,1,1,-1]
                ])

    X = X.T
    X = X.astype(np.float64)

    o = np.array([
                [0],
                [0],
                [0],
                [1],
                [1],
                [1],
                [1],
                [0],
                [0],
                [0],
```

```
            [0],
            [1],
            [0],
            [0],
            [0],
            [1]
            ])

h = np.array([
        [0, 0, 0, 0, 0],
        [0, 0, 0, 0, 0],
        [0, 0, 0, 0, 0],
        [0, 1.0, 0, 0, 0],
        [0, 0, 1.0, 1.0, 0],
        [0, 0, 1.0, 0, 0],
        [0, 0, 0, 1.0, 0],
        [0, 0, 0, 0, 0],
        [0, 0, 0, 0, 0],
        [0, 0, 0, 0, 0],
        [0, 0, 0, 0, 0],
        [1.0, 1.0, 0, 0, 0],
        [0, 0, 0, 0, 0],
        [0, 0, 0, 0, 0],
        [0, 0, 0, 0, 0],
        [1.0, 0, 0, 0, 0],
        ])

W_in = np.array([
                  [0.4, 0, 0.4, 0.4, 1],
                  [0, -0.6, 0.6, 0.6, 1],
                  [-0.4, 1.2, -0.4, 0, 1],
                  [-0.4, 1.2, 0, -0.4, 1]
                  ])


W_h = np.array([
                [1],
                [1],
                [1],
                [1],
                [0.5]
                  ])

W_in_up = np.array([
                [1, 0, 1, 1, 2.5],
                [0, -1, 1, 1, 1.5],
                [-1, 1, -1, 0, 0.5],
```

```
                    [-1, 1, 0, -1, 0.5]
    ])

    trial = 1000
    result = 0
    result_1 = 0
    result_2 = 0

    for i in range(trial):
        result += question_2_accuracy(X,W_in,W_h,o);
        noise = np.random.normal(0, 0.1, size = (5, 16))
        result_1 += question_2_accuracy( (X + noise) , W_in, W_h, o)
        result_2 += question_2_accuracy( (X + noise), W_in_up, W_h, o)

    result /= trial
    result_1 /= trial
    result_2 /= trial

    print("Question 2.b\nAccuracy (mean of 100 trials): ", result, "%")
    print("Question 2.c\nAccuracy w/ noise applied to inputs (mean of 100␣
↪trials) ", result_1, "%")
    print("Question 2.c\nAccuracy after weights changed (mean of 100 trials):",␣
↪result_2, "%")


    ## 2.d

    X = np.array([[0,0,0,0],
                  [0,0,0,1],
                  [0,0,1,0],
                  [0,0,1,1],
                  [0,1,0,0],
                  [0,1,0,1],
                  [0,1,1,0],
                  [0,1,1,1],
                  [1,0,0,0],
                  [1,0,0,1],
                  [1,0,1,0],
                  [1,0,1,1],
                  [1,1,0,0],
                  [1,1,0,1],
                  [1,1,1,0],
                  [1,1,1,1]])

    d = np.array([
              [0],
              [0],
```

```
            [0],
            [1],
            [1],
            [1],
            [1],
            [0],
            [0],
            [0],
            [0],
            [1],
            [0],
            [0],
            [0],
            [1]
            ])

    X = X.astype(np.float64)
    X = np.tile(X, (25,1))
    d = np.tile(d, (25,1))
    theta = np.full((400,1), -1)
    for i in range(100):
        noise = np.random.normal(0, 0.2, size = (X.shape[0], X.shape[1]))
        X_n = np.c_[X + noise, theta]
        result_1 += question_2_accuracy(X_n.T , W_in, W_h, d)
        result_2 += question_2_accuracy(X_n.T, W_in_up, W_h, d)

    print("Question 2.d\nAccuracy of 2.a (mean of 100 trials): ", result_1/100,␣
↪"%")
    print("Accuracy of 2.c (mean of 100 trials)", result_2/100, "%")
```

[5]:
```
def question_2_accuracy(X, W_in, W_h, d, update = False):
    temp = W_in @ X
    temp = temp.T
    assert(temp.shape == (d.shape[0],4))
    h = np.c_[temp, np.full((d.shape[0],1), -1)]
    assert(h.shape == (d.shape[0],5))
    h = np.heaviside(h, 0)
    o = h @ W_h
    o = np.heaviside(o, 0)
    return (d == o).mean() * 100
```

[6]:
```
question_2()
```

```
Question 2.b
Accuracy (mean of 100 trials):  100.0 %
Question 2.c
Accuracy w/ noise applied to inputs (mean of 100 trials)  92.19375 %
```

```
Question 2.c
Accuracy after weights changed (mean of 100 trials): 97.33125 %
Question 2.d
Accuracy of 2.a (mean of 100 trials):  86.1044375 %
Accuracy of 2.c (mean of 100 trials) 90.54081249999999 %
```

3 Question 3

[7]:
```
def sigmoid(v ,lam = 1, polarity = 0):
    if polarity == 0:
        return 1 / (1 + np.exp(- v * lam))
    else:
        return (1 - np.exp(- v * lam)) / (1 + np.exp(- v * lam))
```

[8]:
```
def corr_plot(ims):

    #flattens images
    ims = np.reshape(ims, (ims.shape[0], ims.shape[1] * ims.shape[2]))
    #chose 26 images, first of their class
    corr_ims = ims[0:-1:200]
    corr = np.corrcoef(corr_ims)

    f, axarr = plt.subplots(figsize=(15, 15), dpi=80)
    im = axarr.imshow(corr)

    for i in range(26):
        for j in range(26):
            text = axarr.text(j, i, float(np.round(corr[i,j], 2)),
                              ha="center", va="center", color="w")

    axarr.set_title("Correlation Coefficient Matrix between 26 images (one from␣
↪each class)")
    Classes = list(string.ascii_lowercase)
    plt.xticks(range(26), Classes)
    plt.yticks(range(26), Classes)
    plt.colorbar(im);
    plt.savefig("CORR_MATRIX.png")
```

[9]:
```
def disp(d = None, w = False, mse = False, mse_lol = None, learning_rate = None):
    if mse == True:

        plt.plot(mse_lol[1], 'r', label=('\u03B7 =' + str(learning_rate[1])))
        plt.plot(mse_lol[2], 'b', label=('\u03B7 =' + str(learning_rate[2])))
        plt.plot(mse_lol[0], 'g', label=('\u03B7 =' + str(learning_rate[0])))
        plt.legend(loc='best')
        plt.xlabel('Sample Size')
```

```
        plt.ylabel('MSE')
        plt.title("MSE for different \u03B7 values")
        plt.savefig("MSE_SAMEPLOT.png")

        plt.figure()
        f, axarr = plt.subplots(3, figsize=(10,7))
        f.suptitle("MSE for different \u03B7 values")
        axarr[0].plot(mse_lol[0], 'g')
        axarr[0].set_title('\u03B7 =' + str(learning_rate[0]))
        axarr[1].plot(mse_lol[1], 'r')
        axarr[1].set_title('\u03B7 =' + str(learning_rate[1]))
        axarr[2].plot(mse_lol[2], 'b')
        axarr[2].set_title('\u03B7 =' + str(learning_rate[2]))
        plt.savefig("MSE_SUBPLOTS.png")

    else:
        f, axarr = plt.subplots(5,6, figsize=(10,12))

        k = 0

        for i in range(5):
            for j in range(6):
                if k < 26:
                    if w == True:
                        axarr[i][j].imshow(d[k])
                        title = "WEIGHT_VISUAL.png"
                    else:
                        axarr[i][j].imshow(d[200*k].T)
                        title = "SAMPLE_VISUAL.png"
                    axarr[i][j].set_title("Class: " + str(k + 1))
                axarr[i][j].axis('off')
                k += 1
        f.suptitle(title)
        plt.savefig(title)
```

[10]:
```
def question_3_init(ims_shape):

    mse_list = []

    W = np.random.normal(0, 0.01, size = (ims_shape[1] * ims_shape[2], 26))
    b = np.random.normal(0, 0.01, size = (1,1))
    o = np.zeros((26,1))
    dw = np.zeros((ims_shape[1] * ims_shape[2], 26))
    db = np.zeros((1,1))

    return W, b, o, dw, db, mse_list
```

[11]:
```
def question_3_propagate(ims, lbls, W, b, o, dw, db, mse_list, learning_rate = 0.
↪5, epoch = 10000, lam = 1):

    for i in range(epoch):

        k = np.random.randint(0, ims.shape[0])

        X = np.reshape(ims[k].T, (ims.shape[1] * ims.shape[2], 1))
        X = question_3_normalize(X)

        d = np.full((26,1), 0)
        d[int(lbls[k]) - 1] = 1

        o = sigmoid(W.T @ X - b, lam, 0)

        mse = np.sum((o - d) ** 2)/26
        mse_list.append(mse)

        ds =  (d - o) * lam * o * (1 - o)
        # ds =  (d - o) * (lam / 2.0) * (1 - o * o) #bipolar sigmoid

        dW = learning_rate * ( X @ ds.T)
        db = learning_rate * ds * -1

        W = W + dW
        b = b + db

    return mse_list, W, b
```

[12]:
```
def question_3_predict(ims, d, W, b, lam):
    ims = question_3_normalize(ims)
    o = sigmoid(ims @ W - b.T, lam)
    return (np.argmax(o, axis=1) == (d - 1)).mean() * 100
```

[13]:
```
def question_3_normalize(ims):
    if ims.shape[0] > 1 and ims.shape[1] > 1:
        n = ims.max(axis = 1)
        ims *= 1.0/np.reshape(n, (-1,1))
    else:
        ims *= 1.0/ims.max()
    return ims
```

[14]:
```
def question_3(filename = "assign1_data1.h5"):


    h5 = h5py.File(filename,'r')
    #train data
```

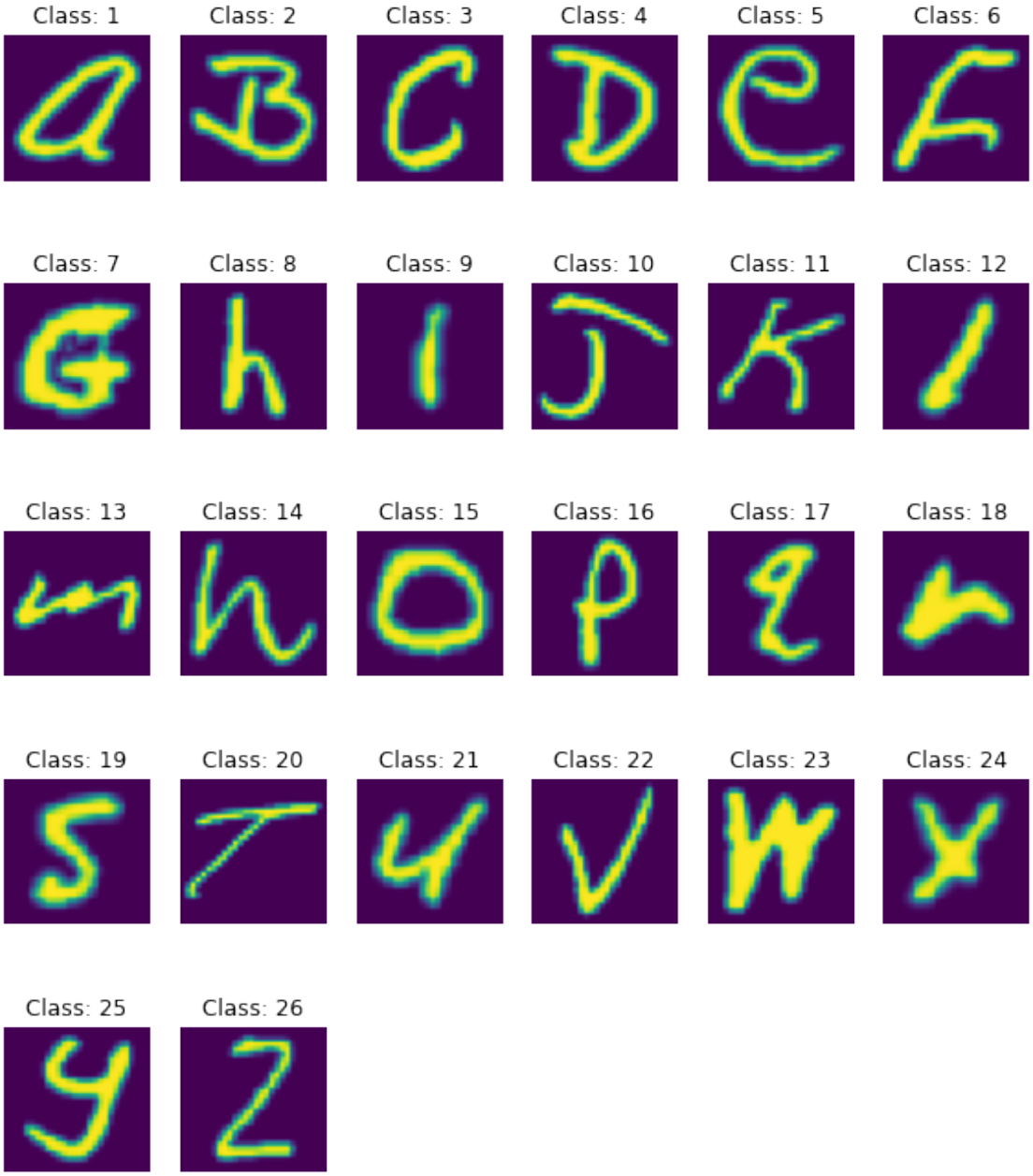
```
    trainims = h5['trainims'][()].astype('float64')
    trainlbls = h5['trainlbls'][()].astype('float64')
    #test data
    testims = h5['testims'][()].astype('float64')
    testlbls = h5['testlbls'][()].astype('float64')
    h5.close()

    #display 26 samples and their correlation coefficient matrix
    disp(trainims)
    corr_plot(trainims.transpose(0,2,1))

    mse_lol = [] #list of lists

    ###TRAINING###

    #determine variables
    ims = trainims
    lbls = trainlbls
    epoch = 10000
    learning_rate = [0.146, 10, 0.0001]
    lam = 1
    a_list = []

    for eta in learning_rate:
        #initialize
        W, b, o, dw, db, mse_list = question_3_init(ims.shape)

        #train (both forward and back propagation)
        mse_list, W, b = question_3_propagate(ims, lbls, W, b, o, dw, db,␣
↪mse_list, eta, epoch, lam)
        mse_lol.append(mse_list)

        #display the final weights, should resemble the actual class shapes
        if eta == learning_rate[0]:
            disp( np.reshape( W.T, (-1, ims.shape[1], ims.shape[2])) , True)

        #TEST: accuracy calculations

        # ##TRAIN ACCURACY
        t = trainims.transpose(0, 2, 1)
        t = np.reshape(t, (t.shape[0], t.shape[1] * t.shape[2]))
        tl = trainlbls

        a = question_3_predict(t, tl, W, b, lam)


        # #TEST ACCURACY
```

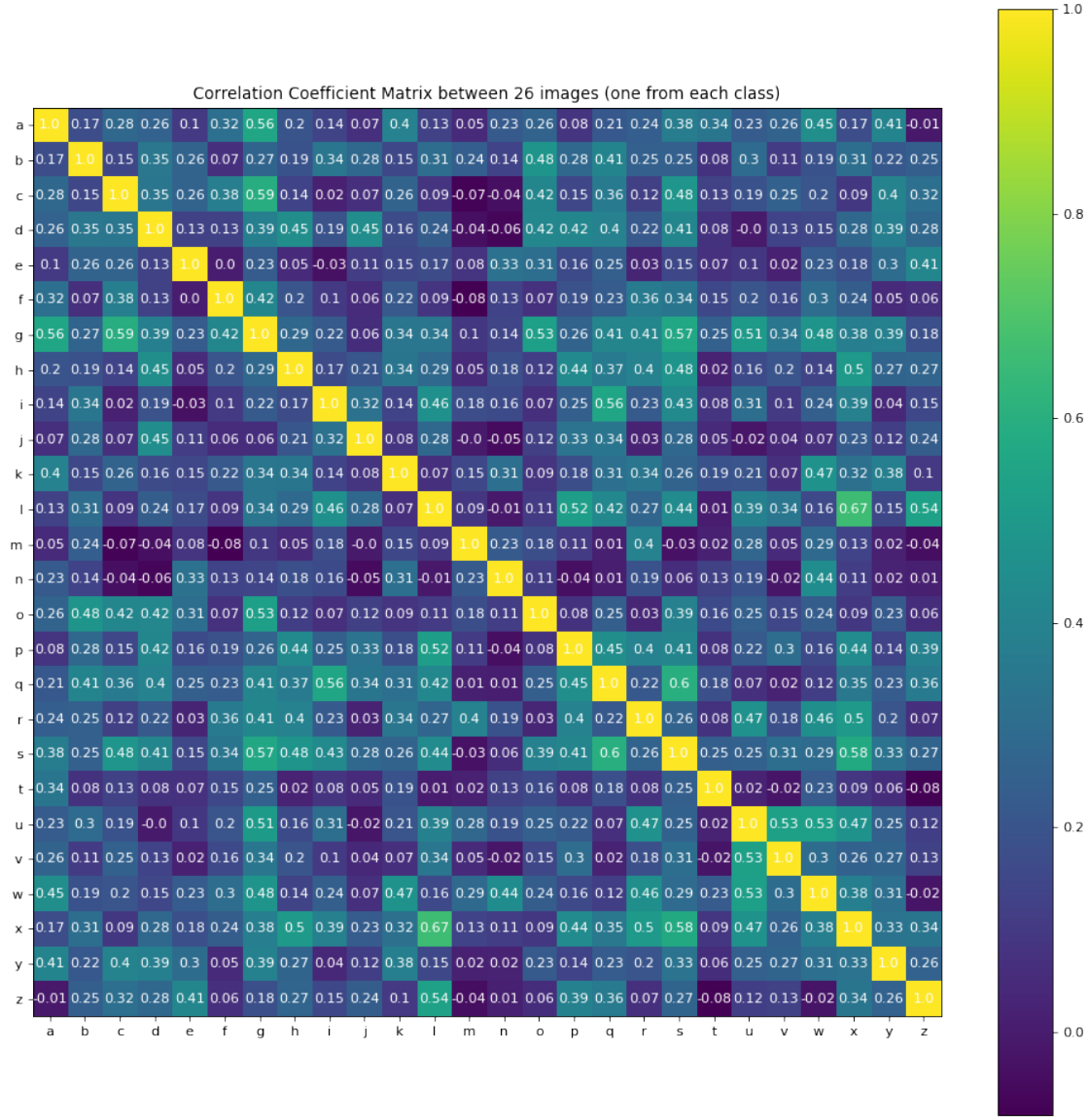
```
        t = testims.transpose(0, 2, 1)
        t = np.reshape(t, (t.shape[0], t.shape[1] * t.shape[2]))
        tl = testlbls
        b = question_3_predict(t, tl, W, b, lam)
        # a_list.append(a)

        print("Results for learning rate = " + str(eta))
        print("Train accuracy: ", a, " %")
        print("Test accuracy: ", b, " %")


    plt.figure(figsize=(10,5))
    disp(mse = True, mse_lol = mse_lol, learning_rate = learning_rate)
```

[15]:
```
question_3()
```

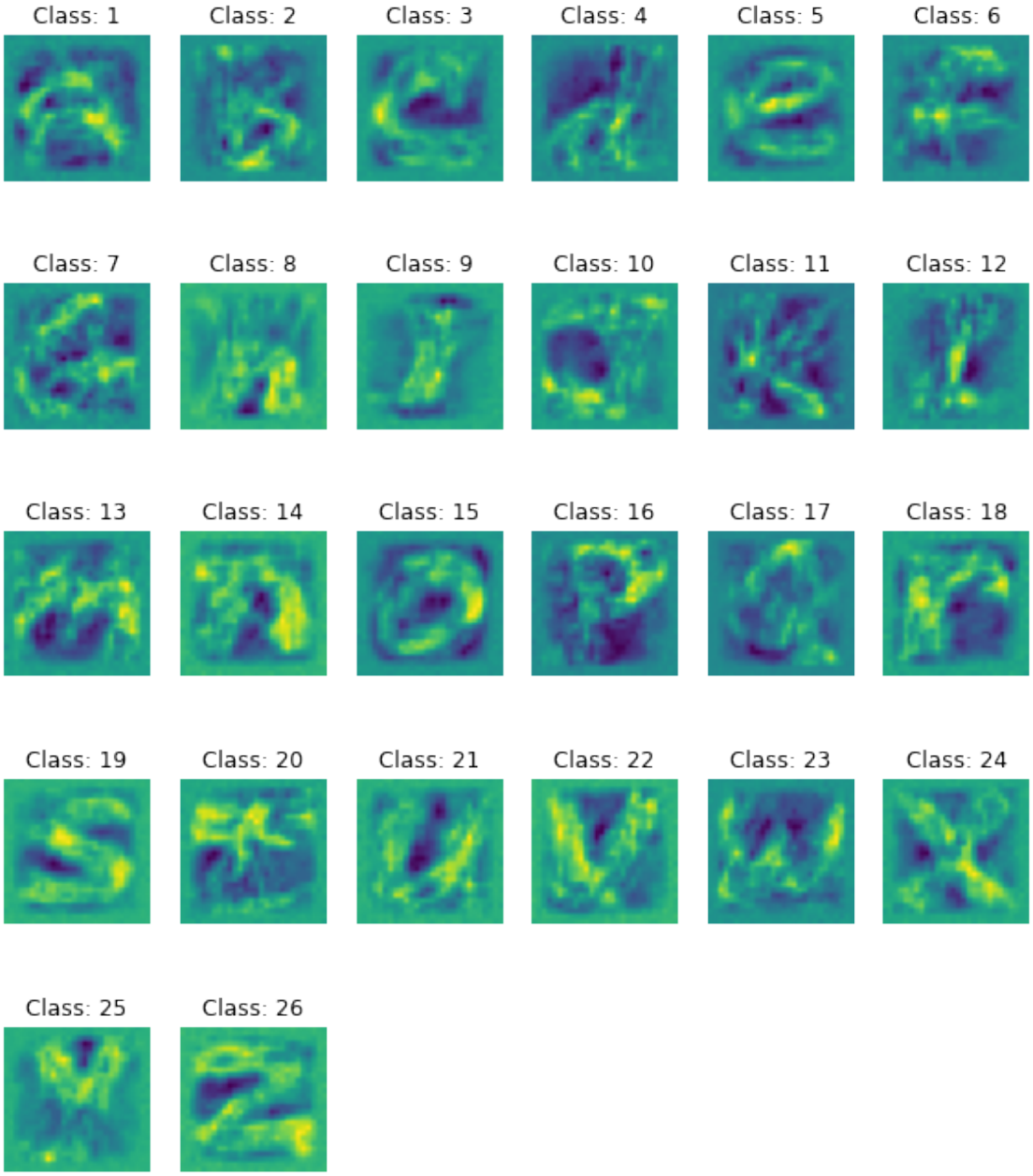
```
Results for learning rate = 0.146
Train accuracy:  64.34615384615384  %
Test accuracy:  59.692307692307686  %
Results for learning rate = 10
Train accuracy:  6.5769230769230775  %
Test accuracy:  6.846153846153847  %
Results for learning rate = 0.0001
Train accuracy:  19.73076923076923  %
Test accuracy:  19.153846153846153  %
```

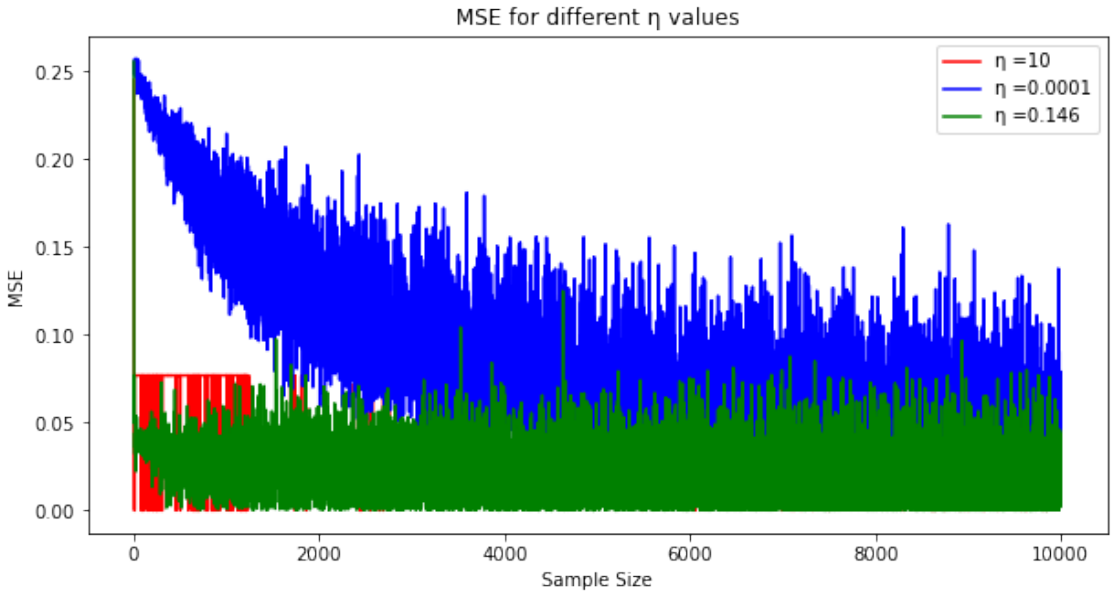
SAMPLE_VISUAL.png



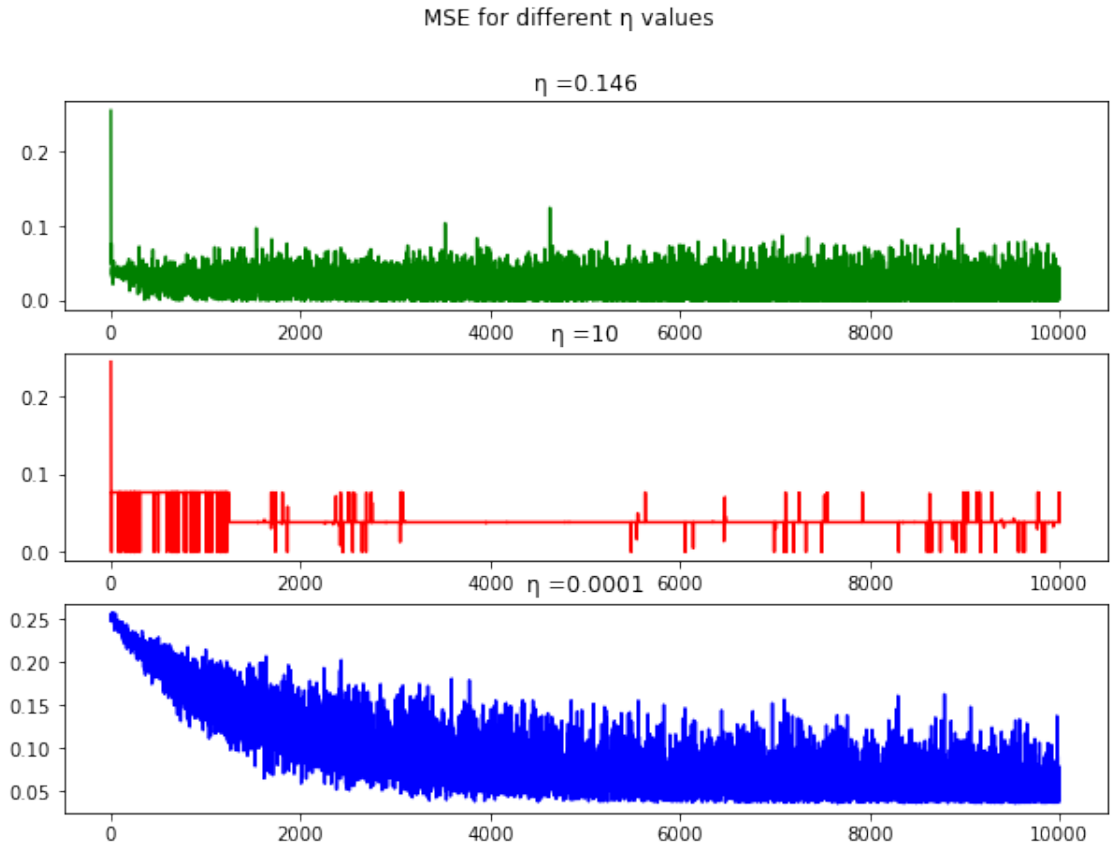

Correlation Coefficient Matrix between 26 images (one from each class)
a 1.0 0.17 0.28 0.26 0.1 0.32 0.56 0.2 0.14 0.07 0.4 0.13 0.05 0.23 0.26 0.08 0.21 0.24 0.38 0.34 0.23 0.26 0.45 0.17 0.41 -0.01
b 0.17 1.0 0.15 0.35 0.26 0.07 0.27 0.19 0.34 0.28 0.15 0.31 0.24 0.14 0.48 0.28 0.41 0.25 0.25 0.08 0.3 0.11 0.19 0.31 0.22 0.25
c 0.28 0.15 1.0 0.35 0.26 0.38 0.59 0.14 0.02 0.07 0.26 0.09 -0.07 -0.04 0.42 0.15 0.36 0.12 0.48 0.13 0.19 0.25 0.2 0.09 0.4 0.32
d 0.26 0.35 0.35 1.0 0.13 0.13 0.39 0.45 0.19 0.45 0.16 0.24 -0.04 -0.06 0.42 0.42 0.4 0.22 0.41 0.08 -0.0 0.13 0.15 0.28 0.39 0.28
e 0.1 0.26 0.26 0.13 1.0 0.0 0.23 0.05 -0.03 0.11 0.15 0.17 0.08 0.33 0.31 0.16 0.25 0.03 0.15 0.07 0.1 0.02 0.23 0.18 0.3 0.41
f 0.32 0.07 0.38 0.13 0.0 1.0 0.42 0.2 0.1 0.06 0.22 0.09 -0.08 0.13 0.07 0.19 0.23 0.36 0.34 0.15 0.2 0.16 0.3 0.24 0.05 0.06
g 0.56 0.27 0.59 0.39 0.23 0.42 1.0 0.29 0.22 0.06 0.34 0.34 0.1 0.14 0.53 0.26 0.41 0.41 0.57 0.25 0.51 0.34 0.48 0.38 0.39 0.18
h 0.2 0.19 0.14 0.45 0.05 0.2 0.29 1.0 0.17 0.21 0.34 0.29 0.05 0.18 0.12 0.44 0.37 0.4 0.48 0.02 0.16 0.2 0.14 0.5 0.27 0.27
i 0.14 0.34 0.02 0.19 -0.03 0.1 0.22 0.17 1.0 0.32 0.14 0.46 0.18 0.16 0.07 0.25 0.56 0.23 0.43 0.08 0.31 0.1 0.24 0.39 0.04 0.15
j 0.07 0.28 0.07 0.45 0.11 0.06 0.06 0.21 0.32 1.0 0.08 0.28 -0.0 -0.05 0.12 0.33 0.34 0.03 0.28 0.05 -0.02 0.04 0.07 0.23 0.12 0.24
k 0.4 0.15 0.26 0.16 0.15 0.22 0.34 0.34 0.14 0.08 1.0 0.07 0.15 0.31 0.09 0.18 0.31 0.34 0.26 0.19 0.21 0.07 0.47 0.32 0.38 0.1
l 0.13 0.31 0.09 0.24 0.17 0.09 0.34 0.29 0.46 0.28 0.07 1.0 0.09 -0.01 0.11 0.52 0.42 0.27 0.44 0.01 0.39 0.34 0.16 0.67 0.15 0.54
m 0.05 0.24 -0.07 -0.04 0.08 -0.08 0.1 0.05 0.18 -0.0 0.15 0.09 1.0 0.23 0.18 0.11 0.01 0.4 -0.03 0.02 0.28 0.05 0.29 0.13 0.02 -0.04
n 0.23 0.14 -0.04 -0.06 0.33 0.13 0.14 0.18 0.16 -0.05 0.31 -0.01 0.23 1.0 0.11 -0.04 0.01 0.19 0.06 0.13 0.19 -0.02 0.44 0.11 0.02 0.01
o 0.26 0.48 0.42 0.42 0.31 0.07 0.53 0.12 0.07 0.12 0.09 0.11 0.18 0.11 1.0 0.08 0.25 0.03 0.39 0.16 0.25 0.15 0.24 0.09 0.23 0.06
p 0.08 0.28 0.15 0.42 0.16 0.19 0.26 0.44 0.25 0.33 0.18 0.52 0.11 -0.04 0.08 1.0 0.45 0.4 0.41 0.08 0.22 0.3 0.16 0.44 0.14 0.39
q 0.21 0.41 0.36 0.4 0.25 0.23 0.41 0.37 0.56 0.34 0.31 0.42 0.01 0.01 0.25 0.45 1.0 0.22 0.6 0.18 0.07 0.02 0.12 0.35 0.23 0.36
r 0.24 0.25 0.12 0.22 0.03 0.36 0.41 0.4 0.23 0.03 0.34 0.27 0.4 0.19 0.03 0.4 0.22 1.0 0.26 0.08 0.47 0.18 0.46 0.5 0.2 0.07
s 0.38 0.25 0.48 0.41 0.15 0.34 0.57 0.48 0.43 0.28 0.26 0.44 -0.03 0.06 0.39 0.41 0.6 0.26 1.0 0.25 0.25 0.31 0.29 0.58 0.33 0.27
t 0.34 0.08 0.13 0.08 0.07 0.15 0.25 0.02 0.08 0.05 0.19 0.01 0.02 0.13 0.16 0.08 0.18 0.08 0.25 1.0 0.02 -0.02 0.23 0.09 0.06 -0.08
u 0.23 0.3 0.19 -0.0 0.1 0.2 0.51 0.16 0.31 -0.02 0.21 0.39 0.28 0.19 0.25 0.22 0.07 0.47 0.25 0.02 1.0 0.53 0.53 0.47 0.25 0.12
v 0.26 0.11 0.25 0.13 0.02 0.16 0.34 0.2 0.1 0.04 0.07 0.34 0.05 -0.02 0.15 0.3 0.02 0.18 0.31 -0.02 0.53 1.0 0.3 0.26 0.27 0.13
w 0.45 0.19 0.2 0.15 0.23 0.3 0.48 0.14 0.24 0.07 0.47 0.16 0.29 0.44 0.24 0.16 0.12 0.46 0.29 0.23 0.53 0.3 1.0 0.38 0.31 -0.02
x 0.17 0.31 0.09 0.28 0.18 0.24 0.38 0.5 0.39 0.23 0.32 0.67 0.13 0.11 0.09 0.44 0.35 0.5 0.58 0.09 0.47 0.26 0.38 1.0 0.33 0.34
y 0.41 0.22 0.4 0.39 0.3 0.05 0.39 0.27 0.04 0.12 0.38 0.15 0.02 0.02 0.23 0.14 0.23 0.2 0.33 0.06 0.25 0.27 0.31 0.33 1.0 0.26
z -0.01 0.25 0.32 0.28 0.41 0.06 0.18 0.27 0.15 0.24 0.1 0.54 -0.04 0.01 0.06 0.39 0.36 0.07 0.27 -0.08 0.12 0.13 -0.02 0.34 0.26 1.0
a b c d e f g h i j k l m n o p q r s t u v w x y z
1.0
0.8
0.6
0.4
0.2
0.0

WEIGHT_VISUAL.png





<Figure size 432x288 with 0 Axes>



Question 4

The following pages in this subsection contain the Jupyter Notebook that was commented on and ran in Question 4

1 Implementing a Neural Network

In this exercise we will develop a neural network with fully-connected layers to perform classification, and test it out on the CIFAR-10 dataset.

[1]:
```
# A bit of setup

import numpy as np
import matplotlib.pyplot as plt

from cs231n.classifiers.neural_net import TwoLayerNet

from __future__ import print_function

%matplotlib inline
plt.rcParams['figure.figsize'] = (10.0, 8.0) # set default size of plots
plt.rcParams['image.interpolation'] = 'nearest'
plt.rcParams['image.cmap'] = 'gray'

# for auto-reloading external modules
# see http://stackoverflow.com/questions/1907993/autoreload-of-modules-in-ipython
%load_ext autoreload
%autoreload 2

def rel_error(x, y):
    """ returns relative error """
    return np.max(np.abs(x - y) / (np.maximum(1e-8, np.abs(x) + np.abs(y))))
```

We will use the class `TwoLayerNet` in the file `cs231n/classifiers/neural_net.py` to represent instances of our network. The network parameters are stored in the instance variable `self.params` where keys are string parameter names and values are numpy arrays. Below, we initialize toy data and a toy model that we will use to develop your implementation.

[2]:
```
# Create a small net and some toy data to check your implementations.
# Note that we set the random seed for repeatable experiments.

input_size = 4
hidden_size = 10
num_classes = 3
num_inputs = 5

def init_toy_model():
    np.random.seed(0)
    return TwoLayerNet(input_size, hidden_size, num_classes, std=1e-1)

def init_toy_data():
    np.random.seed(1)
    X = 10 * np.random.randn(num_inputs, input_size)
```

```
    y = np.array([0, 1, 2, 2, 1])
    return X, y

net = init_toy_model()
X, y = init_toy_data()
```

2 Forward pass: compute scores

Open the file `cs231n/classifiers/neural_net.py` and look at the method `TwoLayerNet.loss`. This function is very similar to the loss functions you have written for the SVM and Softmax exercises: It takes the data and weights and computes the class scores, the loss, and the gradients on the parameters.

Implement the first part of the forward pass which uses the weights and biases to compute the scores for all inputs.

[3]:
```
scores = net.loss(X)
print('Your scores:')
print(scores)
print()
print('correct scores:')
correct_scores = np.asarray([
  [-0.81233741, -1.27654624, -0.70335995],
  [-0.17129677, -1.18803311, -0.47310444],
  [-0.51590475, -1.01354314, -0.8504215 ],
  [-0.15419291, -0.48629638, -0.52901952],
  [-0.00618733, -0.12435261, -0.15226949]])
print(correct_scores)
print()

# The difference should be very small. We get < 1e-7
print('Difference between your scores and correct scores:')
print(np.sum(np.abs(scores - correct_scores)))
```

```
Your scores:
[[-0.81233741 -1.27654624 -0.70335995]
 [-0.17129677 -1.18803311 -0.47310444]
 [-0.51590475 -1.01354314 -0.8504215 ]
 [-0.15419291 -0.48629638 -0.52901952]
 [-0.00618733 -0.12435261 -0.15226949]]

correct scores:
[[-0.81233741 -1.27654624 -0.70335995]
 [-0.17129677 -1.18803311 -0.47310444]
 [-0.51590475 -1.01354314 -0.8504215 ]
 [-0.15419291 -0.48629638 -0.52901952]
 [-0.00618733 -0.12435261 -0.15226949]]
```

```
Difference between your scores and correct scores:
3.6802720496109664e-08
```

3 Forward pass: compute loss

In the same function, implement the second part that computes the data and regularizaion loss.

[4]:
```
loss, _ = net.loss(X, y, reg=0.05)
correct_loss = 1.30378789133

# should be very small, we get < 1e-12
print('Difference between your loss and correct loss:')
print(np.sum(np.abs(loss - correct_loss)))
```

```
Difference between your loss and correct loss:
1.7985612998927536e-13
```

4 Backward pass

Implement the rest of the function. This will compute the gradient of the loss with respect to the variables `W1`, `b1`, `W2`, and `b2`. Now that you (hopefully!) have a correctly implemented forward pass, you can debug your backward pass using a numeric gradient check:

[5]:
```
from cs231n.gradient_check import eval_numerical_gradient

# Use numeric gradient checking to check your implementation of the backward␣
 ↪pass.
# If your implementation is correct, the difference between the numeric and
# analytic gradients should be less than 1e-8 for each of W1, W2, b1, and b2.

loss, grads = net.loss(X, y, reg=0.05)

# these should all be less than 1e-8 or so
for param_name in grads:
    f = lambda W: net.loss(X, y, reg=0.05)[0]
    param_grad_num = eval_numerical_gradient(f, net.params[param_name],␣
 ↪verbose=False)
    print('%s max relative error: %e' % (param_name, rel_error(param_grad_num,␣
 ↪grads[param_name])))
```

```
W1 max relative error: 3.561318e-09
b1 max relative error: 1.555470e-09
W2 max relative error: 3.440708e-09
b2 max relative error: 3.865091e-11
```

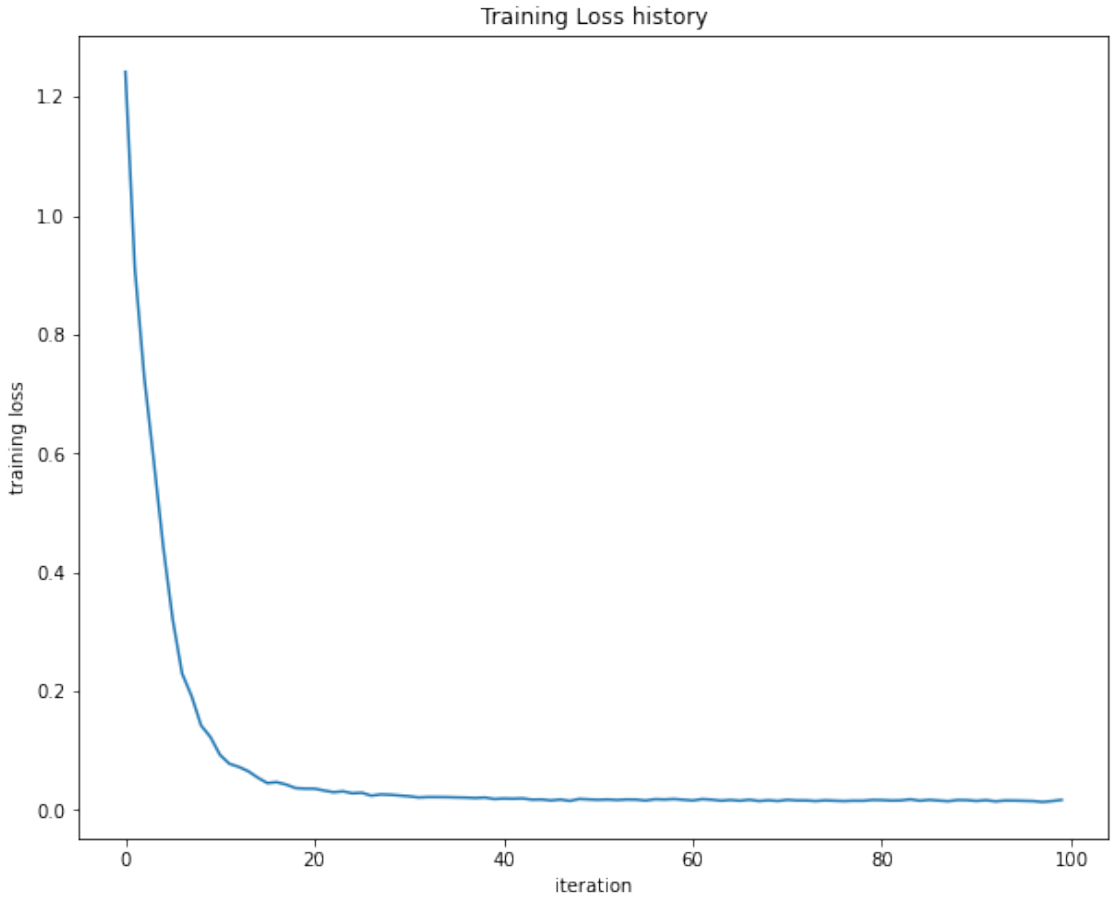
5 Train the network

To train the network we will use stochastic gradient descent (SGD), similar to the SVM and Softmax classifiers. Look at the function `TwoLayerNet.train` and fill in the missing sections to implement the training procedure. This should be very similar to the training procedure you used for the SVM and Softmax classifiers. You will also have to implement `TwoLayerNet.predict`, as the training process periodically performs prediction to keep track of accuracy over time while the network trains.

Once you have implemented the method, run the code below to train a two-layer network on toy data. You should achieve a training loss less than 0.2.

[6]:
```
net = init_toy_model()
stats = net.train(X, y, X, y,
            learning_rate=1e-1, reg=5e-6,
            num_iters=100, verbose=False)

print('Final training loss: ', stats['loss_history'][-1])

# plot the loss history
plt.plot(stats['loss_history'])
plt.xlabel('iteration')
plt.ylabel('training loss')
plt.title('Training Loss history')
plt.show()
```

```
Final training loss:  0.017149607938732093
```



6 Load the data

Now that you have implemented a two-layer network that passes gradient checks and works on toy data, it's time to load up our favorite CIFAR-10 data so we can use it to train a classifier on a real dataset.

[7]:
```
from cs231n.data_utils import load_CIFAR10

def get_CIFAR10_data(num_training=49000, num_validation=1000, num_test=1000):
    """
    Load the CIFAR-10 dataset from disk and perform preprocessing to prepare
    it for the two-layer neural net classifier. These are the same steps as
    we used for the SVM, but condensed to a single function.
    """
    # Load the raw CIFAR-10 data
    cifar10_dir = 'cs231n/datasets/cifar-10-batches-py'

    X_train, y_train, X_test, y_test = load_CIFAR10(cifar10_dir)
```

```
    # Subsample the data
    mask = list(range(num_training, num_training + num_validation))
    X_val = X_train[mask]
    y_val = y_train[mask]
    mask = list(range(num_training))
    X_train = X_train[mask]
    y_train = y_train[mask]
    mask = list(range(num_test))
    X_test = X_test[mask]
    y_test = y_test[mask]

    # Normalize the data: subtract the mean image
    mean_image = np.mean(X_train, axis=0)
    X_train -= mean_image
    X_val -= mean_image
    X_test -= mean_image

    # Reshape data to rows
    X_train = X_train.reshape(num_training, -1)
    X_val = X_val.reshape(num_validation, -1)
    X_test = X_test.reshape(num_test, -1)

    return X_train, y_train, X_val, y_val, X_test, y_test


# Cleaning up variables to prevent loading data multiple times (which may cause␣
 ↪memory issue)
try:
   del X_train, y_train
   del X_test, y_test
   print('Clear previously loaded data.')
except:
   pass

# Invoke the above function to get our data.
X_train, y_train, X_val, y_val, X_test, y_test = get_CIFAR10_data()
print('Train data shape: ', X_train.shape)
print('Train labels shape: ', y_train.shape)
print('Validation data shape: ', X_val.shape)
print('Validation labels shape: ', y_val.shape)
print('Test data shape: ', X_test.shape)
print('Test labels shape: ', y_test.shape)
```

```
Train data shape:  (49000, 3072)
Train labels shape:  (49000,)
Validation data shape:  (1000, 3072)
```

```
Validation labels shape:  (1000,)
Test data shape:  (1000, 3072)
Test labels shape:  (1000,)
```

7 Train a network

To train our network we will use SGD. In addition, we will adjust the learning rate with an exponential learning rate schedule as optimization proceeds; after each epoch, we will reduce the learning rate by multiplying it by a decay rate.

[8]:
```
input_size = 32 * 32 * 3
hidden_size = 50
num_classes = 10
net = TwoLayerNet(input_size, hidden_size, num_classes)

# Train the network
stats = net.train(X_train, y_train, X_val, y_val,
            num_iters=1000, batch_size=200,
            learning_rate=1e-4, learning_rate_decay=0.95,
            reg=0.25, verbose=True)

# Predict on the validation set
val_acc = (net.predict(X_val) == y_val).mean()
print('Validation accuracy: ', val_acc)
```

```
iteration 0 / 1000: loss 2.302954
iteration 100 / 1000: loss 2.302550
iteration 200 / 1000: loss 2.297648
iteration 300 / 1000: loss 2.259602
iteration 400 / 1000: loss 2.204170
iteration 500 / 1000: loss 2.118565
iteration 600 / 1000: loss 2.051535
iteration 700 / 1000: loss 1.988466
iteration 800 / 1000: loss 2.006591
iteration 900 / 1000: loss 1.951473
Validation accuracy:  0.287
```

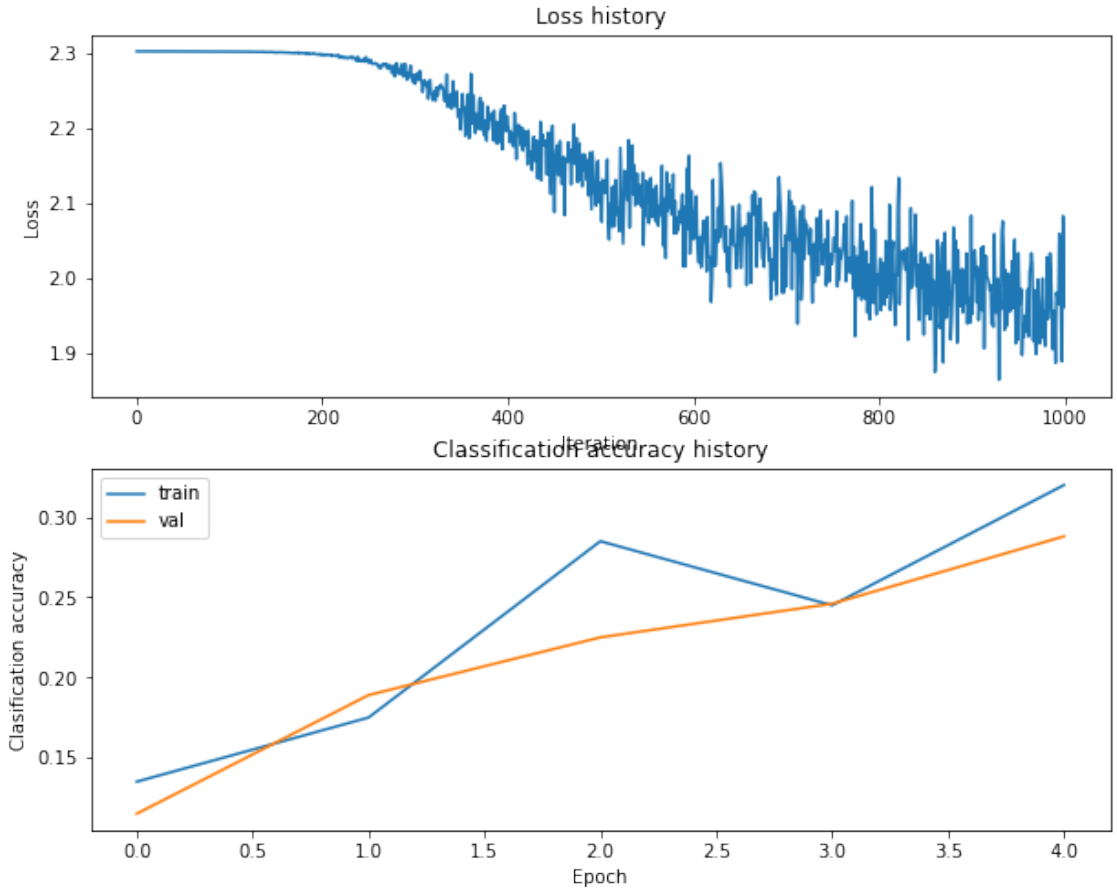
8 Debug the training

With the default parameters we provided above, you should get a validation accuracy of about 0.29 on the validation set. This isn't very good.

One strategy for getting insight into what's wrong is to plot the loss function and the accuracies on the training and validation sets during optimization.

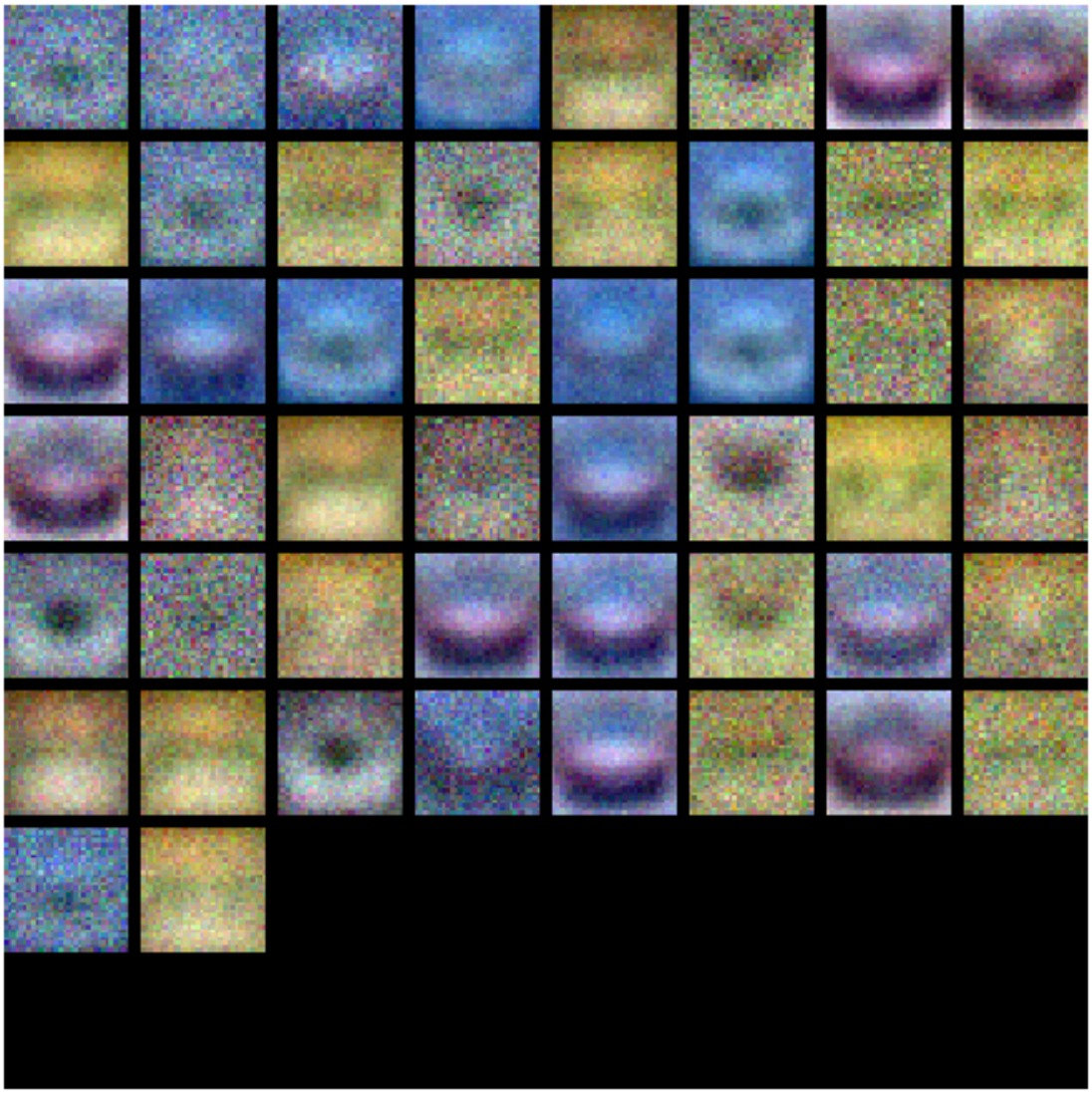
Another strategy is to visualize the weights that were learned in the first layer of the network. In most neural networks trained on visual data, the first layer weights typically show some visible structure when visualized.

[9]:
```
# Plot the loss function and train / validation accuracies
plt.subplot(2, 1, 1)
plt.plot(stats['loss_history'])
plt.title('Loss history')
plt.xlabel('Iteration')
plt.ylabel('Loss')

plt.subplot(2, 1, 2)
plt.plot(stats['train_acc_history'], label='train')
plt.plot(stats['val_acc_history'], label='val')
plt.title('Classification accuracy history')
plt.xlabel('Epoch')
plt.ylabel('Clasification accuracy')
plt.legend()
plt.show()
```



[10]:
```
from cs231n.vis_utils import visualize_grid

# Visualize the weights of the network
```

```
def show_net_weights(net):
    W1 = net.params['W1']
    W1 = W1.reshape(32, 32, 3, -1).transpose(3, 0, 1, 2)
    plt.imshow(visualize_grid(W1, padding=3).astype('uint8'))
    plt.gca().axis('off')
    plt.show()

show_net_weights(net)
```



9 Tune your hyperparameters

What's wrong?. Looking at the visualizations above, we see that the loss is decreasing more or less linearly, which seems to suggest that the learning rate may be too low. Moreover, there is no gap between the training and validation accuracy, suggesting that the model we used has low capacity, and that we should increase its size. On the other hand, with a very large model we would expect to see more overfitting, which would manifest itself as a very large gap between the training and validation accuracy.

Tuning. Tuning the hyperparameters and developing intuition for how they affect the final performance is a large part of using Neural Networks, so we want you to get a lot of practice. Below, you should experiment with different values of the various hyperparameters, including hidden layer size, learning rate, numer of training epochs, and regularization strength. You might also consider tuning the learning rate decay, but you should be able to get good performance using the default value.

Approximate results. You should be aim to achieve a classification accuracy of greater than 48% on the validation set. Our best network gets over 52% on the validation set.

Experiment: You goal in this exercise is to get as good of a result on CIFAR-10 as you can, with a fully-connected Neural Network. Feel free implement your own techniques (e.g. PCA to reduce dimensionality, or adding dropout, or adding features to the solver, etc.).

[24]:
```
best_net = None # store the best model into this

################################################################################
# TODO: Tune hyperparameters using the validation set. Store your best trained  #
# model in best_net.                                                            #
#                                                                               #
# To help debug your network, it may help to use visualizations similar to the  #
# ones we used above; these visualizations will have significant qualitative    #
# differences from the ones we saw above for the poorly tuned network.          #
#                                                                               #
# Tweaking hyperparameters by hand can be fun, but you might find it useful to  #
# write code to sweep through possible combinations of hyperparameters          #
# automatically like we did on the previous exercises.                          #
################################################################################

input_size = X_train.shape[1]
hidden_size = 75
output_size = 10

# learning_rates = [1, 1e-1, 1e-2, 1e-3]
# regularization_strengths = [1e-6, 1e-5, 1e-4, 1e-3]

# Magic lrs and regs?
# Just copy from: https://github.com/lightaime/cs231n/blob/master/assignment1/
 ↪two_layer_net.ipynb
# :(
```

```
learning_rates = np.array([1.5,2,2.5])*1e-3
regularization_strengths = [0.5, 0.55,0.62, 0.65]

best_val = -1

for lr in learning_rates:
    for reg in regularization_strengths:
        net = TwoLayerNet(input_size, hidden_size, output_size)
        net.train(X_train, y_train, X_val, y_val, learning_rate=lr, reg=reg,
                  num_iters=1500)

        y_val_pred = net.predict(X_val)
        val_acc = np.mean(y_val_pred == y_val)

        print('lr: %f, reg: %f, val_acc: %f' % (lr, reg, val_acc))

        if val_acc > best_val:
            best_val = val_acc
            best_net = net

print('Best validation accuracy: %f' % best_val)
################################################################################
#                                END OF YOUR CODE                              #
################################################################################
```

```
lr: 0.001500, reg: 0.500000, val_acc: 0.463000
lr: 0.001500, reg: 0.550000, val_acc: 0.490000
lr: 0.001500, reg: 0.620000, val_acc: 0.482000
lr: 0.001500, reg: 0.650000, val_acc: 0.472000
lr: 0.002000, reg: 0.500000, val_acc: 0.480000
lr: 0.002000, reg: 0.550000, val_acc: 0.472000
lr: 0.002000, reg: 0.620000, val_acc: 0.470000
lr: 0.002000, reg: 0.650000, val_acc: 0.458000
lr: 0.002500, reg: 0.500000, val_acc: 0.424000
lr: 0.002500, reg: 0.550000, val_acc: 0.474000
lr: 0.002500, reg: 0.620000, val_acc: 0.443000
lr: 0.002500, reg: 0.650000, val_acc: 0.468000
Best validation accuracy: 0.490000
```

[25]:
```
# visualize the weights of the best network
show_net_weights(best_net)
```

10 Run on the test set

When you are done experimenting, you should evaluate your final trained network on the test set; you should get above 48%.

[26]:
```
test_acc = (best_net.predict(X_test) == y_test).mean()
print('Test accuracy: ', test_acc)
```

```
Test accuracy:  0.506
```

Inline Question

Now that you have trained a Neural Network classifier, you may find that your testing accuracy is much lower than the training accuracy. In what ways can we decrease this gap? Select all that

apply. 1. Train on a larger dataset. 2. Add more hidden units. 3. Increase the regularization strength. 4. None of the above.

Your answer: 1, 2, 3

Your explanation:

As a disclaimer, all options given are situational. In some cases training on a larger dataset and increasing the amount of hidden units may cause overfitting, and a high regularization strength may cause underfitting/slow learning, these are situations which increase the gap between test accuracy and train accuracy. However, there will be cases where a larger dataset, increasing the amount of hidden units or/and the regularization strength will yield better accuracy for the test. Below are simple explanations as to why.

1 - Training on a larger dataset means the network will be more adaptable to changes in test data (assuming the larger samples are varied). Hence training a larger data set means that the test accuracy will increase.

2 - Adding more hidden units will increase the test accuracy since it will create more weights that cover a larger set of class inputs.

3 - Regulization strength is used to prevent overfitting, and if the neural network is suspected to overfit, increasing this parameter will yield bettter results.